Strategies for Using Proximal Policy Optimization in Mobile Puzzle Games

Jeppe Theiss Kristensen
IT University of Copenhagen/Tactile Games
Copenhagen, Denmark
jetk@itu.dk

Paolo Burelli
IT University of Copenhagen/Tactile Games
Copenhagen, Denmark
pabu@itu.dk

***Abstract*—While traditionally a labour intensive task, the testing of game content is progressively becoming more automated. Among the many directions in which this automation is taking shape, automatic play-testing is one of the most promising thanks also to advancements of many supervised and reinforcement learning (RL) algorithms. However these type of algorithms, while extremely powerful, often suffer in production environments due to issues with reliability and transparency in their training and usage.**

In this research work we are investigating and evaluating strategies to apply the popular RL method Proximal Policy Optimization (PPO) in a casual mobile puzzle game with a specific focus on improving its reliability in training and generalization during game playing.

We have implemented and tested a number of different strategies against a real-world mobile puzzle game (Lily's Garden from Tactile Games). We isolated the conditions that lead to a failure in either training or generalization during testing and we identified a few strategies to ensure a more stable behaviour of the algorithm in this game genre.

***Index Terms*—reinforcement learning, ppo, player agent, player modelling, playtesting, autonomous agent**

Fig. 1. Level 11 from Lily's Garden. The left hand side shows number of moves left to finish the level, and the board pieces below indicate which and how many pieces to collect before completing the level. Collecting the objectives is done by clearing them off the board, which can be done by clicking on two or more basic pieces of the same color, or using power pieces that clear an entire row/area. Power pieces can be created by matching 5 or more basic pieces.

I. Introduction

Human game testing is an expensive and slow process. It usually requires the full attention of the testers, and there are limitations on how fast humans can operate. Game developers are therefore increasingly starting to use automated play testing. However, developing and implementing such methods in practice has its problems – the methods tend to require a very specific setup for one game, and trying to adapt it to other environments may sometimes break the algorithm and render it useless. In this paper we therefore set out to explore both novel and common strategies for ensuring a stable implementation of a reinforcement learning (RL) play-testing agent in a mobile puzzle game in a production setting.

A popular choice for creating play testing tools is reinforcement learning, and research in this field is moving fast. Novel algorithms and updates to current state-of-the-art methods are constantly being introduced in the latest publications, showing better performance on typical frameworks such as the Arcade Learning Environment [3].

However, contrary to these kind of one-shot evaluations, adapting these methods in a production environment in a company requires additional considerations – such as ease-of-use and long-term reliability. Unlike these benchmark games, production games are updated frequently, and it can not be expected to be possible to draw on expert knowledge at any time in case something goes wrong. Until more focus has been put on strategies on *how* to use these methods, adoption of these methods in the industry will be slow at best.

In this research work we focus on the challenges of implementing the popular RL method Proximal Policy Optimization (PPO) [29], a widely used algorithm available in various RL libraries (OpenAI Baselines/stable-baselines [6], [14], TF-Agents [30], Unity ML-Agents [17]), in a mobile puzzle game called Lily's Garden by Tactile Games (Fig. 1). While other RL methods may also work in this environment, we choose to only focus on PPO since it is one of the two main algorithms implemented in Unity ML-Agents and thus widely accessible to game developers that use Unity.

Our contribution is two-fold:

- We explore different setups for training an agent in a mobile puzzle game and determine a set of hyperparameters and setups that enable the agent to some extend play both seen and unseen levels competently.
- We highlight that the impact of some PPO variations are not fully understood and can easily lead to unexpected learning behaviours. We then suggest strategies for avoid-

ing such behaviours and ensure a more stable training.

This paper is structured as follows: First we introduce the game environment that we will use for testing. Next we present the basics of the PPO algorithm and discuss the specific implementation we use. This is followed by the experiments section where we present the various setups we tested and highlight the main difficulties and problems encountered during training of the agent. Lastly we discuss which methods and strategies that are feasible to employ in a production setting and identify areas that need improvement.

II. Related work

When it comes to creating agents for playing games, reinforcement learning (RL) and deep learning methods have started to become a staple and have been used to play a large variety of games, ranging from arcade games to first-person shooter games [18].

Each genre has its own challenges, and some approaches work better than others in different settings. It is therefore relevant to consider which approaches that have been used for play-testing in similar game genres.

In Atari games, some of the state-of-the-art approaches using pixel data or memory-features as input are deep Q-learning (DQN) [24], [25] and variations thereof (such as Rainbow [13]), and actor-critic approaches like PPO [29] and soft actor-critic [10] in [4].

The *MuZero* algorithm introduced by Schrittwieser et al. [28] uses a combination of tree-search planning and a learned model of the environment and is capable of playing Go, Chess, Shogi and the Atari games. However, how to deal with stochastic transitions was not examined.

As for approaches used specifically on puzzle games, other approaches have also been directly applied. Gudmundsson et al. [9] treat the task as a classification problem and train a convolutional neural network on player data. Their method beats state-of-the-art Monte-Carlo Tree Search algorithms in terms of difficulty prediction and training time and has been used actively for a year by the time of publication. However, this method requires play-through data which may not always be available. Mugrai et al. [26] use a MCTS method with an evolutionary strategy where the fitness function is used to mimic specialised player personas/strategies with different goals, such as maximising score or minimising moves used. This aspect of creating human-like agents is indeed important if they are to be used as a play-testing tool, which is also highlighted by Zhao et al. [36] where the agents are evaluated by considering both skill and style. A comparison of three popular methods (DQN, PPO and A3C) in a custom match-3 game is done by Kamaldinov et al. [20] which shows that the A3C method achieves the highest accumulative reward while the PPO and DQN methods perform worse than random. They use a custom match-3 environment, though, so it is not clear if these results reflect real-world results in puzzle games. An example of training an agent using actual games levels can be seen in the Unity blogpost [33], where an agent for playing Snoopy Pop using ML-Agents in [33] is attempted using a actor-critic method (SAC, [10]) and imitation learning (GAIL, [15]). Although a slightly different genre, it shows that the training can be sped up using sample efficient methods and a player to guide the agent initially. However, a similar approach is not efficient in games like Lily's Garden since in those cases it is not necessary to simulate physics. Furthermore, there are more than 1500 levels available so deciding which levels to train on or alternatively have a player play through all of them is not scalable.

When it comes to using such automated systems in a production setting, reliability and accessibility of the algorithm are critical components. The less interference required, the better, and when something goes wrong, identifying the points of failure easily is important so it can be fixed quickly and not waste resources. RL systems, especially PPO approaches, tend to be the antithesis of these requirements: they tend to be brittle [11], and the stability tends to be implementation-dependant [16]. It is therefore important to consider not just the algorithms but also the strategies of how a play-testing tool should be developed.

Such a tool also needs to be able to generalise to new levels, and one problem that appears in many RL papers is overfitting to an environment [35]. Ways to diagnose and improve the generalisation in deep RL systems have been examined is various works [27], [35]. Farebrother et al. [7] find that dropout and $\ell 2$ regularisation with a DQN method improve generalisation. This is also supported by the findings by Cobbe et al. [5] where data augmentation, batch normalisation and stochasticity were also found to improve generalisation in an implementation of PPO. Adding entropy regularisation also helps find smoother solutions but is very environment dependent [1]. Variations in the levels by using procedural content generation methods can also improve generalisation and help learn more difficult levels [19]. Avoiding undesirable and dangerous actions may also help the agent learn more efficiently because of better and safer exploration strategies [34]. Having the system learn which actions to eliminate has been the focus in some recent works [2], [31], [34]. In addition to learning action blocking, Kenton et al. [21] also use an ensemble model in both a DQN and PPO setup. While the DQN method showed improvements, the PPO experiments showed little improvement compared to the baseline.

III. Environment

In this paper we focus on one game, Lily's Garden[1]. It is a free to play casual puzzle mobile game where you progress through the main story by completing levels. The main gameplay is matching similar colored pieces and thereby collecting objectives (collectgoals), which must be done before running out of moves. The game board has a maximum size of 13 by 9, and in each position, board pieces with various attributes may be placed. The basic pieces can be destroyed/collected if two or more of the same color are next to each other and will create power pieces if 5 or more are

[1] Android, Apple

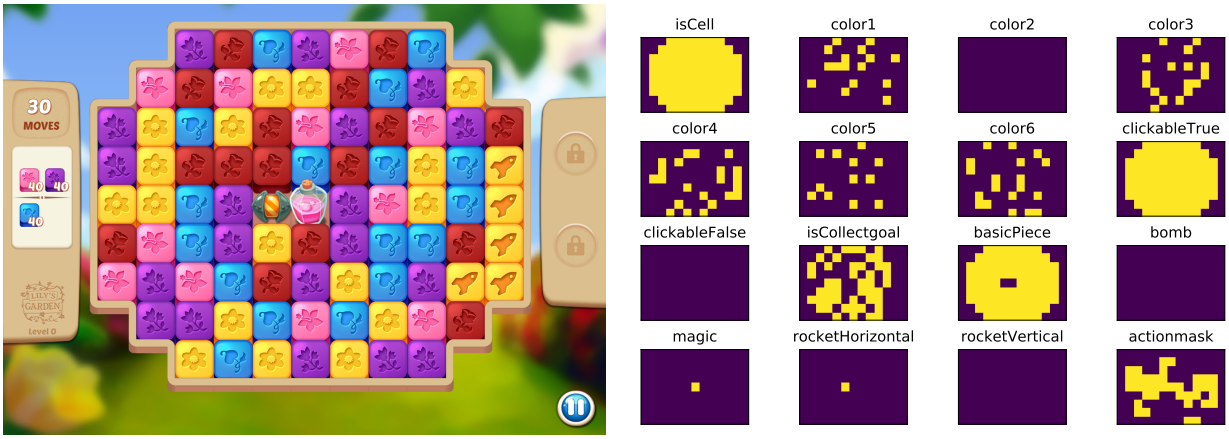

Fig. 2. Example of how an in-game level looks like and how the game board is represented using different channels corresponding to certain board piece attributes. Note that the last channel, the action mask, is only included in certain experiments.

next to each other. The power pieces can be clicked at any time and destroy everything in for example a line or circle around the position. Lastly there are unclickable board pieces, or blockers, that can be removed by matching basic pieces next to it or sometimes only by using a power piece. An example of a level is shown in Fig. 1.

We set up an OpenAI gym environment that connects to headless version of the game (no graphical interface) which, for speed purposes, allows us to play through levels without rendering any graphics. We define a rich reward function where the reward is calculated at each step as: $r = c_{\text{collection}} n + c_{\text{completion}} - 0.1$, where $c_{\text{collection}} = 0.05$, n is the number of collected collectgoals and $c_{\text{completion}} = 1$ if all collectgoals have been collected. The negative term, -0.1, is added to encourage the agent finishing the level faster as to not get a large negative accumulated reward. Given that a typical level has around 50 collectables and requires up to 25 moves to complete, the expected final reward is $R \approx 50 \cdot 0.05 - 25 \cdot 0.1 + 1 = 1$ (not considering discount).

Since each board piece may be of the same type (e.g. basic or blocker) but different attributes (e.g. color or gravity), one-hot encoding each board piece by the unique combination of attributes may lead to a very large and sparse representation, as seen in [9]. Instead we choose to represent the observation space by using layers that correspond to the attributes of all the board pieces in a given position (see Fig. 2). Specifically, we represent the following attributes with a layer giving a total of 15 channels:

- ISCELL: used to define shape of game board
- COLOR: one-hot encoding of 6 unique colors
- ISCOLLECTGOAL: if board piece is a collectgoal
- ISCLICKABLE[TRUE/FALSE]: one layer for clickable, another for non-clickable since a non-clickable piece may be on top of another
- ID: one-hot encoding of BASICPIECE, ROCKETHORIZONTAL, ROCKETVERTICAL, BOMB and MAGIC

This approach also has an advantage when it comes to generalizability for future versions because the observation space will not depend on graphics updates and new types of board pieces are typically made up of a combination of existing attributes. The action space consists of $9 \times 13 = 117$ discrete actions, corresponding to each square of the game board.

IV. METHODS

The typical reinforcement learning problem consists of an agent that interacts with an environment and receives a reward depending on the action. This loop may then continue indefinitely or until the episode ends. The main purpose of the algorithm is then to learn a behaviour that maximises the accumulated reward [32].

In the original form, PPO refers to a family of policy gradient methods that optimize a (clipped) surrogate objective function using multiple minibatch updates per data sample. However, the exact implementations in various libraries may be slightly different because of other additions such as value scaling or batch normalisation [16]. Common for them all is the suggested function to optimise, which is the sum of several loss functions and is given by

$$L^{CLIP+VF+S}(\theta)_t = \hat{\mathbb{E}}_t \left[L^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](\theta) \right], \tag{1}$$

where $L^{CLIP}(\theta)$ is the clipped surrogate objective function, $L_t^{VF}(\theta)$ is the value function squared-error loss, S is an entropy bonus and c_1 and c_2 are coefficients. The $L^{CLIP}(\theta)$ term ensures that the policy updates will not be too large, and the $L_t^{VF}(\theta)$ term is to ensure that the loss from both policy and value functions of the neural networks are accounted for. The S entropy term encourages a more random policy (i.e. more exploration) so a larger entropy coefficient c_2 will encourage more exploration.

A. Implementation

Since we want to investigate strategies for implementing PPO in a production environment, we choose to go with a widely used code library. Some of the notable RL libraries are

OpenAI Baselines[2] and Unity ML-Agents[3]. For the following experiments we choose to use stable-baselines, which is a fork from OpenAI Baselines but follows the same algorithmic implementation of PPO.

We test out three different strategies which will be described in this section. These strategies are:

- Color shuffling (CS)
- Resetting
- Action mask

Color shuffling refers to swapping the color channels in the observations randomly. While color shuffling is done in the post-training evaluation for all models to simulate how levels are designed, we want to test how effective it is to also include this strategy during training. It should also help prevent overfitting – even though it is random which board pieces that drop down and replace cleared pieces, the initial setup are usually predetermined (see Fig. 4) which may lead to strong overfitting.

Resetting the environment commonly happens at the end of episodic environments, which in this case could be when the level is completed or failed. However, the level move limit is subject to change because of design considerations, and we already add a penalty at each step to encourage it finishing faster. Imposing a move limit does therefore not make much sense. What we do try with the reset strategy, though, is imposing a total step limit, which includes both valid and invalid moves. The reasoning behind this strategy, similar what is given in [23] using restarts in Angry Birds, is that the agent is prevented from exploring useless states that it will not learn anything from.

Before deciding what the maximum episode length could be, two things should be considered. One is how the typical PPO implementation samples observations. In our case, we sample 256 observations before training on these minibatches. This means that if we reset after 256 total steps, we may end up with a full minibatch of bad training samples, which is undesirable. Secondly the typical steps required to pass a level is generally around 50. Levels that require more steps are rare since it would be very frustrating for players to almost finish a level but ultimately fail after, say, 100 steps rather than 50. We therefore choose to reset after 100 steps which should ensure at least some good observations and still allow the agent to complete a level.

Using an action mask during training is the last strategy we explore. While we give a penalty for selecting an invalid action, preventing the agents from selecting certain catastrophic or invalid actions may lead to more efficient learning. The question is how this limitation should be implemented. We use two different approaches for creating action masks in the following experiments – a hard and a soft action mask.

With the hard action mask, the invalid actions are completely masked when sampling from the policy distribution. In practice, this is done by adding the mask to the logits of policy distribution, where valid actions have a value of 0 and invalid action a value of $-\infty$. This is slightly different than in the ML-Agents library where a small probability ϵ is added to the action probabilities which prevents ∞ values but also allow invalid actions to be taken, albeit with a very low probability. The way sampling is done in the stable-baselines library is by using a Gumbel-max trick.[4] Specifically, noise following a Gumbel distribution (computed by taking the negative logarithm twice of uniformly distributed noise) is added to the logits which ensures the sampling will follow the underlying probabilities of the actions.

The soft action mask is a kind of forward model of the environment. Specifically we add the action mask to the observation space as an additional channel, as illustrated in the last panel in Fig. 2. The reason for calling this a soft action mask is because it does not directly prevent invalid actions from being taken although it might significantly reduce the probability. The soft action mask model is denoted with V2.

Since the game simulator does not provide a method for getting the action mask, we define it ourselves. It follows the basic rules that an action is valid if at least two BASICPIECES are adjacent and of the same color, or if there is a power piece in the cell. While this is not true for later levels, it is sufficient for the first 11 levels that we test on.

Lastly, we also want to evaluate if it makes a difference to continue training after the learning curves have plateaued since shorter training times allow for quicker iterations and thus easier testing in a production environment. These long-trained models are denoted in the post-training evaluation figures with (late).

V. EXPERIMENTS

We carried out a number of experiments to test the performance of the PPO algorithm in our environment. We used the PPO2 implementation from the Python RL library stable-baselines [14] and a custom CNN policy (Fig. 3).

Each of the experiments are evaluated similarly to [5] where the trained agent is tested on unseen levels in order to evaluate

[2]https://github.com/openai/baselines

[3]https://github.com/Unity-Technologies/ml-agents

[4]https://github.com/hill-a/stable-baselines/blob/a57c80e0636582995d602309d2ea5547c0d58e61/stable_baselines/common/distributions.py#L323

TABLE I
OVERVIEW OF MODELS AND USED ENTROPY COEFFICIENT (EC) AS WELL AS WHICH TRAINING STEP CHECKPOINT USED FOR POST-TRAINING EVALUATION. THE SECTION IN WHICH THE RESULTS OF SAID MODELS ARE ALSO SHOWN. CS: COLOR SHUFFLE.

Model	**EC**	**Step** ($\times 10^6$)	**Section**
Baseline	0	0.35	VI-A
Baseline	0.001	0.20	VI
Baseline	0.01	11	VI-A
CS	0.001	0.20	VI-A
CS	0.01	14	VI-A
CS+reset	0.01	0.35	VI-B
CS+reset+mask	0.01	6.5	VI-C
CS+reset+maskV2	0.01	4.0	VI-C
CS+reset (late)	0.01	14	VI-B
CS+reset+maskV2 (late)	0.01	14	VI-C

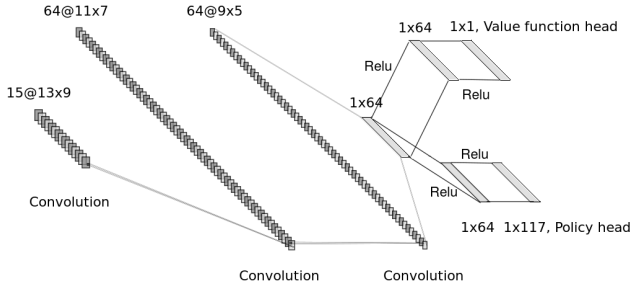

Fig. 3. The network architecture of the agent.

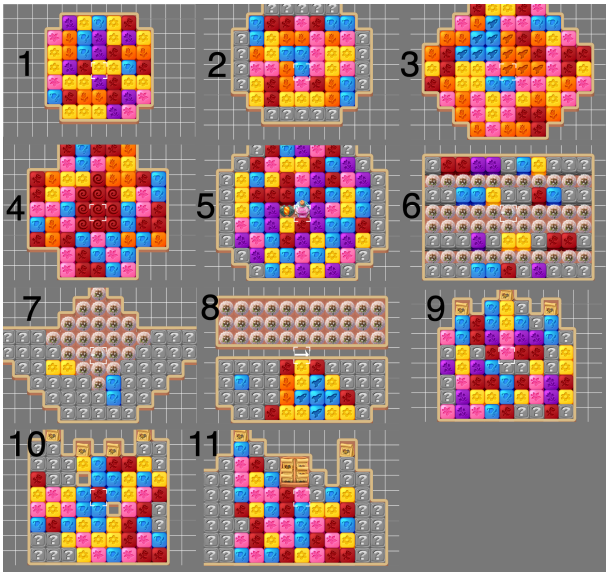

Fig. 4. Levels used for the experiment. Board pieces with question marks are assigned a random color on level start, while every other board piece is hardcoded.

its ability to generalize. This is done by training on 5 chosen levels (1, 3, 5, 7 and 9) selected randomly and uniformly during training and validated using an additional 6 levels (2, 4, 6, 8, 10 and 11). With the exception of level 11, these levels include two unique blockers, and splitting the levels accordingly ensures both that the tutorial levels and trained on and both the training and test sets will include at least one level containing any of the blockers. Level 11 has a third unique blocker so we include that level in the evaluation to see how the agent performs with completely unseen mechanics. An overview of the levels is shown in Fig. 4.

A. Evaluation Metrics

During training we consider the accumulated reward/learning curve as the evaluation statistic. For the post-training evaluation we do not want to only estimate if the agent can finish the level within the actual in-game max moves but also how competent it is compared to a random agent. We therefore allow up to 2000 total steps and do not use an action mask. We also shuffle the colors during evaluation for all models in order to simulate actual in-use performance, since the different colors of the board pieces only affect the aesthetics of the game and are used interchangeably.

We will consider two post-training evaluation metrics:

Competence is the reciprocal average index (starting with 1) of the first valid action after sampling actions using the action probabilities without replacement. Taking the reciprocal value corresponds to estimating the average valid step percentage and can be thought of as a proxy for how well the agent understand the basic match-2 mechanic of the game.

Level completion percentage is calculated by imposing the level move limit on the agent. We also include actual player data. It should be noted that the player completion percentage is estimated by taking the number of level completions over total number of level attempts. However, the level attempts include successes, failures *and* abandoning the game, where the latter may happen if the game for example crashes, other technical failures or simply just giving up on a level. Abandoning the game typically happens less than 5% of the time, though, so this effect should be minor.

B. Model Setup

We did a preliminary analysis training various models with different hyperparameters to find a stable configuration. While we also experimented with reward shaping and state representations, the key changes required to get the PPO algorithm to work with Lily's Garden was changing the minibatch size, number of steps per update and number of actors. We found that setting nminibatches to 64 (default: 4), n_steps to 256 (default: 128) and the number of parallel actors is set to 8 gave a good balance between speed and stability of the algorithm. This is not unexpected as these changes from default ensure a smoother gradient and faster and more stable training [12] and thus more stable training.

We use a custom neural network setup as shown in Fig. 3. It uses three 2x2 convolutions with filter size 64 and leaky relu activations, which are fed into two fully connected 64 layer for the actor and critic heads respectively.

The above hyperparameters are kept the same throughout the experiments except for the entropy coefficient, which will be discussed further in Section VI and VI-A. That setup will serve as a baseline model where no special strategies for training are used. For the other experiments, we use the three aforementioned strategies in Section IV-A.

VI. Results

The learning curves for every model can be seen in Fig. 5, the valid move percentages in Fig. 6 and the completion rate in Fig. 7. Table I shows at which step each model was evaluated as well as in which of the following sections they are discussed further.

In this section, we only consider the baseline model with an entropy coefficient (EC) of 0.001. The other two baseline models are discussed in the next section.

Looking at the learning curve of the Baseline EC: 0.001 model in Fig. 5, it can be seen that the agent quickly learns as reflected in the increase in episode rewards. However,

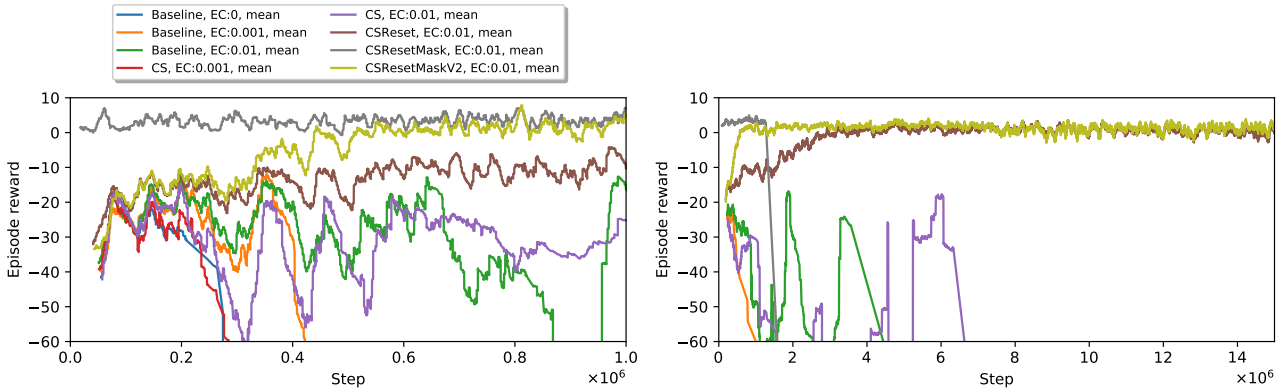

Fig. 5. Learning curves for the tested model. The left figure shows a zoomed in version on the first 1 million steps. CS refers to models trained with color shuffling.

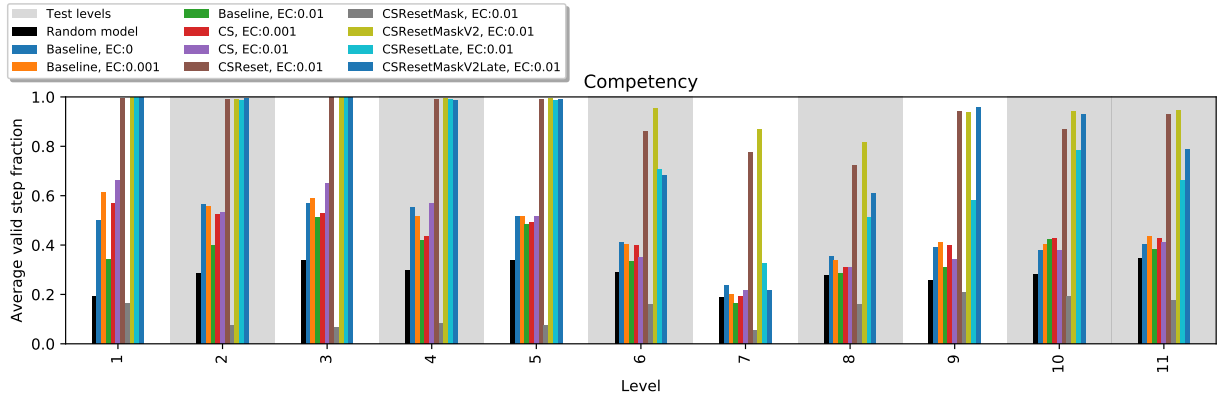

Fig. 6. Number of valid moves per level for each model. CS refers to models trained with color shuffling. The grey shaded levels are the unseen test levels

after 400.000 steps, the episode reward sharply decreases and completely breaks the training. The same behaviour was also observed in other experiments during the initial analysis. This happens when the action entropy becomes sufficiently low which indicates is that the agent ends up picking the same bad action and fills up the training samples with bad observations. The problem is further compounded by the fact that invalid actions do not change the state of the game and we do not do anything to prevent the algorithm from selecting invalid actions, leading to identical training data samples and thus broken learning.

On Fig. 6 and 7 it can be seen that while the agent generally picks valid actions and completes the levels more often than the random agent, it does not reach human-like performance on both seen and unseen levels after level 2.

A. Generalisation

The observation that the agent does not reach human level performance and sometimes also get stuck on an invalid move may indicate that the agent does not explore sufficiently. One way to increase exploration with a PPO algorithm is to increase the entropy coefficient which adds an entropy bonus to the loss function (c_2 in Eq. (1)). Three different configurations were tested: 0.0, 0.001 and 0.01, where 0.001 is the default value.

Generally the learning curves are very similar but the higher the entropy coefficient is, the longer the agent can be trained for and the less likely it is to encounter catastrophic learning behaviours.

Adding color shuffling should also help the agents generalise because it adds randomness. Indeed, the completion percentage for the CS, EC:0.01 model on level 3 and 4 is better than any of the baseline models and comparable on the other levels. While it should be noted that the model had been training for longer, this was made possible because of the higher entropy coeifficient and more environment stochasticity. Color shuffling therefore seems to be a viable strategy in addition to a high entropy coefficient.

B. Max Episode Length

Using strategies that add randomness and increase exploration are not enough to prevent the agent from sampling the same move over and over again as evidenced by the previous experiments. We therefore try the strategy of resetting the

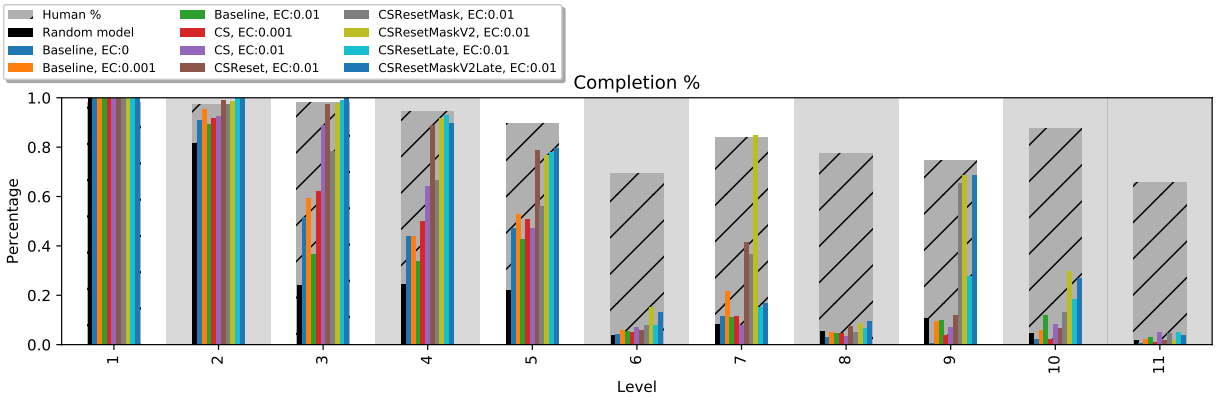

Fig. 7. Completion rate for each model given the level limit. CS refers to models trained with color shuffling. The unseen test levels are shown with a grey shaded area as in 6, while the human completion rate is shown as the large grey hatched bar.

environment to break the loop if a bad learning behaviour happens.

It should be noted that it is difficult to compare the learning curves of agents trained with reset and those without, since resetting ensures that it is not possible to accumulate large negative rewards from choosing the same invalid action over and over again. However, the learning curves are still useful for verifying that the agent is improving and not encountering catastrophic learning behaviour.

Using the reset strategy has a large positive impact on the learning. None of the agents that employ this strategy run into the same loop of selecting the same action all the time which enables the agent to train longer and learn more, with the exception of the CSResetMask model which will be described in the next section. Resetting the environment after a number of steps is therefore a good strategy that leads to more stable learning.

C. Action Masks

Since none of the other experiments directly prevent invalid actions to be taken, the agent has to first learn to infer which moves that are valid. We therefore test two different ways of adding this information – a hard mask and a soft mask, dubbed V2, as described in Section IV-A.

Using a hard action mask very quickly leads to high rewards which makes sense since invalid actions lead to a -0.5 penalty but are never taken now. However, as far as stability goes, the training completely fails after around 1.5 million steps, as seen in the sharp drop in the learning curve on Fig. 5.

Unlike what was seen in many of the previous experiments when the entropy becomes very low/zero, it now receives undefined rewards, indicating something with the algorithm itself is failing. What is happening is that the action probability distribution from the policy is 100% of an invalid action, and 0 on the rest, but because of the hard action mask, the final logits distribution is filled with $-\infty$. Taking the maximum of this vector then leads to unexpected behaviour. This is supported by the fact that the trained agent is actually not very competent (even worse than random, Fig. 6) and thus tend to select invalid actions first.

The picture is completely different when using it as a soft action mask. Looking at Fig. 5, the CSResetMaskV2 agent is both stable during training and learns faster compared to the CSReset agent (i.e. they reach the same learning plateau after 0.5M and 4M steps, respectively). It also has better completion rate and competency on both test and training levels than any of the other approaches.

VII. Discussion

The most effective strategy for training seems to be resetting the environment after a number of total steps. Color shuffling together with an increased entropy coefficient are also strategies that help the agent learn despite slowing down the training. Shifting towards more exploration and less exploitation in games like Lily's Garden therefore seems to be beneficial.

Some of the strategies did not work very well, though, like using a hard action mask or training for too long. This gives rise to some concerns if used in a production environment and will be discussed below.

A. Dealing With Invalid Actions

The main issue encountered throughout the experiments was invalid actions, which may be very specific to our environment and implementation of PPO. For example, in ML-Agents a small probability ϵ is added to the raw probabilities ensuring that there will be no $-\infty$ when converting to logits. This avoids the hard action mask problem, but it can be argued that it is not a hard action mask anymore. Other ways to deal with sampling the same action over and over could be to use an epsilon-greedy approach or by sampling the way we did it in post-training evaluation but this significantly slows down the training.

While this problem with invalid actions may be a very specific problem to our environment, it still highlights a possible issue that may arise in other similar games where some actions do not progress the game. Additionally, if a

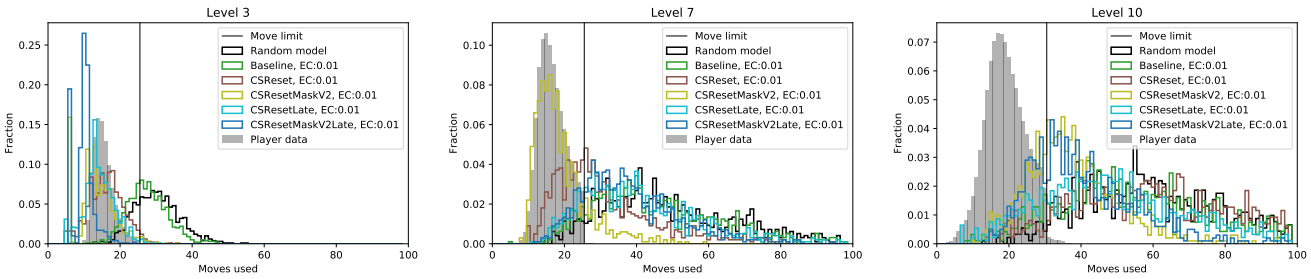

Fig. 8. Histograms of how many moves required to finish selected levels for selected agents, including the random agent in black, that show super-, normal and sub-human performance. The grey shaded area is actual player data and is the distribution that we want to mimic. A sharp cut-off can be seen in the player data distributions which is aligned with the move limit. The reason a small tail can be seen in level 10 is because players are able to purchase an additional 5 moves if they fail, but only a fraction of players choose to do so. The normalisation is therefore also not completely comparable since the agents are allowed to play past the move limit.

hard action mask is being used, the algorithm runs the risk of masking out every action, leading to unexpected behaviour. This is an issue since in a number of research papers on play-testing agents it is not clear how the action masking is actually being done even though it has a huge impact on the training of the agent. This adds further complexity to understanding the algorithms and reflects the thoughts in [16] that the implementation matters.

One question is whether using the action mask is practical in the long run since it does require some kind of modelling of the environment. Additionally, while the levels considered in this paper do not have very complex game mechanics, later levels include mechanics that prevent certain actions. While the environment could be configured to return a proper action, this may prove computationally and developer-time intensive and therefore not viable in the long run. However, interestingly enough it was found out during the evaluation process that the action mask in some very specific cases allowed invalid actions. Whether that is because a bug in the game or action mask modelling is not clear, but it was interesting that this was not a problem when using the soft action mask. This suggest that using even a imperfect forward model of the environment still improves learning.

B. Usefulness in Production

There are two things to consider before judging if the agent is actually useful to level designers.

The first question is whether the level designers would be able to rely on the agent or not. For that to be the case, the more consistent and performant it is, especially on unseen levels, the better. However, what is observed is that the completion rate is worse on unseen levels. This limits the usefulness to level designers since the new levels will obviously not have been encountered before. One solution for this could be to allow the agent to train on the unseen levels. To see whether this is feasible in a production setting requires further testing.

One other thing to take note of is the fact that the completion rate is low despite picking valid action most of the time. This suggests that the agents learn how to play the game but not how to play it optimally. This may be a consequence of the reward function, though – a relatively big penalty is given for selecting invalid moves compared to collecting objectives. The first thing the agents learn is therefore how to not take an invalid action. Learning new things, such as going after objectives, is secondary and would require more training without overfitting. The best way to achieve this would be to introduce more levels, which should help with generalising and making the agent more consistent.

The second thing to consider is that it must play like a human and not superhuman, so the estimated difficulty matches with how players perceive it. While the completion percentage used in the post-training evaluation already reflects this aspect, it does not tell the whole story. Another way to judge how human-like the agent behaves is by looking at the distribution of moves required to finish the level (Fig. 8) and comparing with human data. This kind of visualisation is also more useful to level designers since it can be used to determine the move limit. However, none of the models are consistent in being super/sub-human which must be addressed first.

C. Future Work

When an agent first tries to learn how to play these puzzle games, it first needs to figure out how to do a valid move. As revealed by using an action mask, it learns much faster if something guides it initially. One way could therefore be to use imitation learning to first teach it how to do the basics. This also has the added benefit that it may be easier to guide the agent to play more like a human which would make the tool more useful to level designers. It would require some time and effort to set this up in practice, though, both in regards to implementing it in production code but also the time level designers would have to spend training the agent. Evaluating which approach is more time-effective should therefore not only include computation time but also the human resources required. However, an imitation learning module has been added to ML-Agents and may provide a good starting point.

The post-training evaluations show that the agents play some levels well but struggle with others. It therefore seems like a better strategy to spend more time training on the

difficult levels rather rather than continuing selecting the levels randomly. An idea could be an automated approach like in [8] where the training examples that yield the most learning are chosen. This would also open up for training on more levels which should help generalisation of the agent on unseen levels. One thing to keep in mind before training on many new levels and mechanics, though, is that the agent may be prone to catastrophic forgetting [22] where previously learned behaviours are completely forgotten.

VIII. Conclusion

In this research paper we have successfully adapted the popular RL method PPO to a production grade puzzle game for training play-testing agents. Crucial to this success, not considering hyper-parameter tuning, was introducing a reset strategy where the environment is reset after a fixed number of steps. This ensured a more stable training, enabling the models to learn more. Other strategies also improved other aspects of the training – color shuffling improved generalisability, and introducing an action mask as a partial forward model of the environment in the observation greatly improved training speed, though the latter may not always be feasible in other types of games.

When we experimented with a hard action mask that was added to the logits of the action probabilities, the algorithm completely broke down. This happened because all the valid actions from the model were practically 0 while the invalid actions were all 0 because of the action mask, effectively masking out every action and leading to unexpected behaviour. Various RL libraries use a similar method but it should be used with great caution. A better approach would be to include the action mask in the observations and thus serving as a partial forward model.

IX. Acknowledgements

This work has been supported by the Innovation Fund Denmark and Tactile Games.

We thank Hunter Park (https://github.com/H-Park) and Kenneth Tang (https://github.com/ChengYen-Tang) for discussions on the various implementations.

We also thank Rasmus Berg Palm (ITU) for helpful comments for the manuscript.

References

[1] Zafarali Ahmed, Nicolas Le Roux, Mohammad Norouzi, and Dale Schuurmans. Understanding the impact of entropy on policy optimization. 2018.

[2] Mohammed Alshiekh, Roderick Bloem, Ruediger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe Reinforcement Learning via Shielding. 2017.

[3] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *IJCAI International Joint Conference on Artificial Intelligence*, 2015-Janua:4148–4152, 2015.

[4] Petros Christodoulou. Soft Actor-Critic for Discrete Action Settings. 2019.

[5] Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying Generalization in Reinforcement Learning, 2018.

[6] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. https://github.com/openai/baselines, 2017.

[7] Jesse Farebrother, Marlos C. Machado, and Michael Bowling. Generalization and Regularization in DQN. 2018.

[8] Alex Graves, Marc G. Bellemare, Jacob Menick, Remi Munos, and Koray Kavukcuoglu. Automated Curriculum Learning for Neural Networks. *34th International Conference on Machine Learning, ICML 2017*, 3:2120–2129, 2017.

[9] Stefan Freyr Gudmundsson, Philipp Eisen, Erik Poromaa, Alex Nodet, Sami Purmonen, Bartlomiej Kozakowski, Richard Meurling, and Lele Cao. Human-Like Playtesting with Deep Learning. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2018.

[10] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *35th International Conference on Machine Learning, ICML 2018*, 5:2976–2989, 1 2018.

[11] Perttu Hämäläinen, Amin Babadi, Xiaoxiao Ma, and Jaakko Lehtinen. PPO-CMA: Proximal Policy Optimization with Covariance Matrix Adaptation. 2018.

[12] Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, S. M. Ali Eslami, Martin Riedmiller, and David Silver. Emergence of Locomotion Behaviours in Rich Environments. 2017.

[13] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, pages 3215–3222, 2018.

[14] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

[15] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. *Advances in Neural Information Processing Systems*, pages 4572–4580, 2016.

[16] Andrew Ilyas, Logan Engstrom, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Are Deep Policy Gradient Algorithms Truly Policy Gradient Algorithms? pages 1–40, 11 2018.

[17] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A General Platform for Intelligent Agents. pages 1–18, 2018.

[18] Niels Justesen, Philip Bontrager, Julian Togelius, and Sebastian Risi. Deep Learning for Video Game Playing. *IEEE Transactions on Games*, PP(c):1–1, 2019.

[19] Niels Justesen, Ruben Rodriguez Torrado, Philip Bontrager, Ahmed Khalifa, Julian Togelius, and Sebastian Risi. Illuminating Generalization in Deep Reinforcement Learning through Procedural Level Generation. 6 2018.

[20] Ildar Kamaldinov and Ilya Makarov. Deep Reinforcement Learning in Match-3 Game. In *2019 IEEE Conference on Games (CoG)*, number 4, pages 1–4. IEEE, 8 2019.

[21] Zachary Kenton, Angelos Filos, Owain Evans, and Yarin Gal. Generalizing from a few environments in safety-critical reinforcement learning. pages 1–16, 2019.

[22] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences of the United States of America*, 114(13):3521–3526, 2017.

[23] Tommy Liu, Jochen Renz, Peng Zhang, and Matthew Stephenson. Using Restart Heuristics to Improve Agent Performance in Angry Birds. 5 2019.

[24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. pages 1–9, 12 2013.

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller,

Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2 2015.

[26] Luvneesh Mugrai, Fernando Silva, Christoffer Holmgard, and Julian Togelius. Automated playtesting of matching tile games. *IEEE Conference on Computatonal Intelligence and Games, CIG*, 2019-Augus, 2019.

[27] Charles Packer, Katelyn Gao, Jernej Kos, Philipp Krähenbühl, Vladlen Koltun, and Dawn Song. Assessing Generalization in Deep Reinforcement Learning. 2018.

[28] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. 2019.

[29] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. 2017.

[30] Oscar Ramirez Pablo Castro Ethan Holly Sam Fishman Ke Wang Ekaterina Gonina Neal Wu Efi Kokiopoulou Luciano Sbaiz Jamie Smith Gábor Bartók Jesse Berent Chris Harris Vincent Vanhoucke Eugene Brevdo Sergio Guadarrama, Anoop Korattikara. TF-Agents: A library for reinforcement learning in tensorflow. https://github.com/tensorflow/agents, 2018. [Online; accessed 25-June-2019].

[31] Mathieu Seurin, Philippe Preux, and Olivier Pietquin. I'm sorry Dave, I'm afraid I can't do that" Deep Q-learning from forbidden action. 2019.

[32] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press, 2018.

[33] Erving Teng. Training your agents 7 times faster with ML-Agents - Unity Technologies Blog.

[34] Tom Zahavy, Matan Haroush, Nadav Merlis, Daniel J. Mankowitz, and Shie Mannor. Learn What Not to Learn: Action Elimination with Deep Reinforcement Learning. *Advances in Neural Information Processing Systems*, 2018-Decem(NeurIPS):3562–3573, 9 2018.

[35] Amy Zhang, Harsh Satija, and Joelle Pineau. Decoupling Dynamics and Reward for Transfer Learning. 2018.

[36] Yunqi Zhao, Igor Borovikov, Ahmad Beirami, Jason Rupert, Caedmon Somers, Jesse Harder, Fernando de Mesentier Silva, John Kolen, Jervis Pinto, Reza Pourabolghasem, Harold Chaput, James Pestrak, Mohsen Sardari, Long Lin, Navid Aghdaie, and Kazi Zaman. Winning Isn't Everything: Training Human-Like Agents for Playtesting and Game AI. *arXiv: e-prints*, pages 1–16, 2019.